6.0 Software Quality Assessment

The Software Quality Assessment portion of ASP I presents the results of an investigation into three major areas of software quality: programming conventions, computational efficiency and memory utilization. The following paragraphs discuss the various components which make up a software quality analysis. This is not an exhaustive list, but these sections cover most of the issues that need to be addressed in order to provide an adequate assessment of code quality. Selection of evaluation criteria, like the evaluation itself, is a subjective process dependent upon the experiences and viewpoints of the evaluator. Some factors identified in this section may not be fully applicable to ALARM, and other evaluators may disagree with this assessment.

Details of the software quality assessment procedures can be found in the *SMART Process Description* [1].

In developing a methodology for the assessment of the quality of a software product, there can be no single quality measure. The approach taken in this analysis was to: (1) define a number of software characteristics; (2) subdivide each characteristic into components, each specifying a unique metric for the software characteristic; and (3) analyze the software based on the defined metrics. Analysis results were expressed as a numerical score or a qualitative human judgment. The objective analyses were conducted using CASE tools, while the subjective analyses were completed by human inspection of CASE tool output.

The assessment was performed on ALARM92, the beta version of the model which immediately preceded ALARM 3.0. The substance of the evaluation dealt with elements of style used in writing the ALARM92 code. That style is well defined, and generally followed in both ALARM92 and ALARM 3.0. The majority of the code was not changed between these versions. Consequently, most of the comments presented in this section remain valid for ALARM 3.0.

As a result of the size of ALARM92, and time and resource limitations, the analysis focused on a sample subset of subroutines. The results from this sample should provide a reasonable indication of the construction of the simulation as a whole.

A summary of the assessment is provided in table 6.0-1.

Table 6.0-1 Software Quality Assessment Summary

Subsection	Software Quality Criterion	Subjective Quality
6.1.1	Use of Embedded Comments	Adequate, reasonable
6.1.2	Use of Module Preambles	Acceptable
1.1.3	Source Code Formatting	Excellent
1.1.4	Logical File Processing	Acceptable
1.1.5	Variable Declarations	Acceptable
1.1.6	Programming Logic	Acceptable
1.2.1	Modularity	Very Modular
1.2.2	Algorithm Development	Optimized, easy to read
1.3.1	Global Memory	N/A
1.3.2	Local Memory	N/A

6.1 Programming Conventions

Analysis of the sample subroutine set indicates that, with few exceptions, the code follows the ANSI-77 FORTRAN standard. The subroutines BIODMA and ELEV contain machine architecture-specific declarations, but these declarations are commented as “non-FORTRAN 77" by the programmer.

The developers of ALARM used the convention of providing a descriptive preamble at the beginning of each module. This header is described in subsection 6.1.2 below.

The use of programming conventions is evident, as IMPLICIT declarations are used in ALARM92. This allows the code to be more easily understood and modified.

6.1.1 Use of Embedded Comments

ALARM uses a block style for comments, each containing a description of the code to follow. These blocks normally include the variables used, the logic flow, and the alternatives to decisions made.

Most of the modules analyzed display liberal use of comments. In some cases, there are so many comments, readability of the code is affected. The extensive comments provided in all ALARM92 modules are helpful in understanding the code, and the complexity of the code makes it clear why comments are necessary.

The internal documentation of the ALARM92 code is generally adequate and of reasonable quality; i.e., consistent with the code and well written. However, there are some instances where the documentation is inadequate:

1. In a few cases, technical terms are used carelessly and incorrectly. For example, in the subroutine ELEV, the comment in lines 277-284, referring to the term 'stack,' is inconsistent with the definition of stack in computer science literature.

2. There are many situations where a comment, while not absolutely essential, might aid in understanding complex operations.

3. A similar problem is one in which the comments do not match the code and therefore hamper understanding the purpose of the module.

6.1.2 Use of Module Preambles

Module preambles are blocks of comments at the head of a subroutine which include the name of the module, a short description of the module functionality, input and output parameters, subroutines and functions called, developer point of contact, and a short module update history. In ALARM92, a large percentage of the modules begin with a preamble ranging in size from half a page to as many as three pages. Table 6.1-1 shows the actual statistics for documentation headers for each subroutine and the main program. Note that only 61 percent contain complete information. A few modules had no header or comments. Not all modules include a complete preamble, and some preambles are not current.

Table 6.1-1 Preamble Contents Statistics (ALARM92)

Preamble Contents	Number of Modules	Percentage of Total
No Header	11	9%
Origination and Developer Only	19	16%
Origination, Developer & I/O Parameters	13	11%
Origination, Developer & Purpose	3	3%
Origination, Developer, I/O Parameters & Purpose	73	61%

1.1.3 Source Code Formatting

FORMAT statements are located at the bottom of the source files, thereby not interrupting the program's flow. Declarations and COMMONs are located at the beginning of the file, so that variables are declared before they are used.

ANSI-standard FORTRAN programming requires every statement to begin no sooner than column 7 of each record. Beginning all statements in column 7 produces a flat-looking, difficult to follow source program. Indenting style is individual, but indenting itself is necessary to improve the readability of the program listing. In the modules analyzed, both DO loops and IF blocks are indented to improve readability. Additionally, equal levels of complicated calculations are lined up vertically which allow following them relatively easy.

1.1.4 Logical File Processing

In the sample subset of ALARM92 modules examined, all file I/O units are initialized variables and no reference to specific units by number are encountered. This programming practice makes code easy to modify.

1.1.5 Variable Declarations

There are many considerations in assessing the way variables and arrays are declared and used within a model. ALARM explicitly and consistently declares global variables (in COMMON blocks). ALARM explicitly declares all COMPLEX variables, and many calling parameters and local variables; however, ALARM uses the IMPLICIT convention identified below for declaring most of the local variables.

1. Undeclared Variables: Most modules explicitly declare or initialize variables prior to first use. In ALARM92 this is not a consistent practice, however, since in some modules, assignment of type is made according to the first letter of the variable name. The use of default type assignment and initialization is not considered good programming practice.

2. Variable Naming Conventions: In a program the size of ALARM92, a well-defined convention for naming variables should be established early in the development process, especially when variables are limited in length by a standard such as FORTRAN 77. In general, the variable naming convention in ALARM92 is well designed. A defined convention also facilitates

communications between members of the development team and the development of homogeneous code.

3. Consistency: Each module analyzed appears to have a generally consistent internal style, but still shows the trademark styles of more than one programmer.

4. Implicit Naming: Most modern coding practices require the use of the FORTRAN convention IMPLICIT NONE, which requires the explicit declaration of *all* variables. IMPLICIT NONE is used infrequently but IMPLICIT declarations are used. The ALARM coding convention uses the IMPLICIT statement so that variables beginning with the letters A through H and O through Z are implicitly defined to be REAL*8 (double precision) variables. Variables beginning with the letters I through N are implicitly defined to be INTEGER. COMPLEX variables are all explicitly declared, and many begin with the letter C.

5. Descriptive Naming: By inspection of the variable's name, one could not always gain a full explanation as to its purpose. The header/comments, if any, did not always explain the purposes of each of the variables that are used.

6. COMMON Declarations: All COMMON variables are declared explicitly and correctly. COMMON blocks are declared within each module; however, INCLUDE statements are not used.

1.1.6 Programming Logic

The assessment of the logic used in ALARM answered two major questions: Is the code effective, avoiding typical programming pitfalls? Is the code easily understood? These questions are addressed below. In general, there were no serious problems found with the ALARM92 code; it is written in a straightforward style which is easy to read and follow.

1. Unused Variables, Functions, or Subroutines: No unused variables, functions or subroutines were encountered.

2. Directed Branching and Use of Recursion: Extensive use of unconditional branching (i.e., GO TO statements) is not considered good programming practice because it does not support or facilitate modularity and it contributes to complex code that is very difficult to maintain (“spaghetti code”). Few GO TO statements were encountered within the modules analyzed.

3. Recursion: The use of recursion can make a program difficult to read and maintain. Some algorithms, however, are recursive by design, and to code them non-recursively would unnecessarily complicate the program. The use of recursion was not encountered in ALARM92.

4. Duplication of Constant Definitions: No duplicate local constant definitions were encountered between modules.

5. Compiler-Dependent or Hardware-Dependent Extensions: Two subroutines, BIODMA and ELEV, contain hardware-dependent or compiler-dependent extensions. The use of the INTEGER*2 (“small integer”) declaration is dictated by the format of the raw binary Defense Mapping Agency (DMA) Digital Terrain Elevation Data. While the use of two-byte integers is not preferred, most FORTRAN 77 compilers are VAX/VMS FORTRAN compatible and allow this declaration type.

6. Use of Local Variables: It is much faster in terms of computational efficiency to utilize global variables than to have the compiler generate a local variable when needed. Local variables allow for more compact code, however. In ALARM92, extensive use is made of global COMMON variables, and local variables are only used when necessary. ALARM92 appears to have struck a reasonable balance between use of COMMON and local variables.

7. Subroutine Entry/Exit Points: The use of multiple subroutine ENTRY points and exits (RETURN) are not considered to be good programming practices, particularly in FORTRAN, because there is no obvious distinction between a CALL to an ENTRY point or a CALL to a subroutine. No modules were identified that contained multiple ENTRY points.

1.2 Computational Efficiency

Most of the ALARM92 code examined in the assessment seemed to be reasonably efficient. However, there are several places in the code where modest gains in efficiency could be achieved. For example, in subroutine ELEV, the logic replaces data in a buffer already in use through a shifting operation. This approach could cost a lot of execution time if the number of buffers (NBUFFR) were large (more than about 10). For NBUFFR > 10, the use of a linked list instead of an ordered array is recommended.

1.2.1 Modularity

ALARM is a medium sized program, consisting of one main program and 115 subroutines and functions. The main program is contained in file ALARM and is very modular with called subroutines contained in individual files, as shown in table 1.2-1 below.

Table 1.2-1 ALARM 3.0 Source Code Totals

Source Code Category[a]	Number
Number of Files	116
Number of Main Routines	1
Number of Subroutines	105
Number of Functions	10
Number of Other Types	0
Total Source Size	1,455K Bytes
Total Source Lines of Code (including blanks and comments)	30,846

a. Data refer to the ALARM Model Computer Software Component (CSC) only, not including the Support Utilities CSC.

The ALARM92 code is generally very modular. Each routine performs a single well-defined function. Two minor instances of less-than-perfect modular design include the following:

1. In some cases, a SUBROUTINE has been used in a context where a FUNCTION would have been more conventional.

2. There are a few instances where the code seems to be more fragmented than necessary.

1.2.2 Algorithm Development

In-line code is more efficient than the use of subroutines and functions, particularly for a small program. However, for a large program such as ALARM92, the use of subroutines and functions is essential for several reasons. First, repetitious code segments can be easily replicated by subroutine CALLs or FUNCTIONs, making the code more modular and easier to follow. Secondly, subroutines and functions use less valuable memory than repetitious in-line code. The analysis of selected ALARM92 modules found excellent use of subroutines and functions and little repetitious code segments.

Algorithms are optimized in the code, but not so far as to obscure the algorithm. Time efficiency is optimized in the following way: the radar range equation is incrementally calculated as the program flow is traversed. At several points in the calculation, if the signal to interference ratio exceeds the detection threshold, the non-detect flag is set for the current iteration, and further calculation is skipped.

Each module analyzed appears to be designed to achieve one main goal. To break up code effectively, each module should only be concerned with accomplishing a single primary task. For ease in maintainability, algorithms should be developed in a modular fashion. A complicated equation should not be coded in a single statement with multiple continuation lines. Each element in the calculation should be executed on its own line. The ALARM92 modules analyzed appear to follow this practice, and as a result, they were relatively easy to read and follow. No attempt was made in this effort to verify the correctness or authenticity of the algorithms.

1.3 Memory Utilization

Memory management in a FORTRAN program is very different than in a Pascal or C program. In FORTRAN, little control is given to the programmer. The most effective control the programmer has is by the efficient design and use of COMMON blocks. The programmer must exercise common-sense techniques, such as not defining an overly large array that is not used, or declaring variables within COMMON blocks that are never referenced. The operations that affect memory operations in ALARM92 appear to be very efficiently specified.

Another way the programmer can exercise control over memory management is through the declaration of arrays in a global versus local fashion. By the prudent use of local variables, the programmer can restrict the declaration of arrays with large element sizes until run time. Analysis of ALARM92 indicate this practice is followed to the maximum extent practical.

Reusing variables, while it saves memory, will tend to make a large program confusing to follow. Using EQUIVALENCE, while it also saves memory, is not recommended unless necessary. The developers of ALARM92 refrained from these poor programming practices.

1.3.1 Global Memory

Global memory is variable space that is accessible by all the routines during their operation. This access can be hampered by incorrect COMMON or EQUIVALENCE statements. Incorrectly specified variables can affect the available memory for invocation of subprograms.

No occurrences of incorrect COMMON or EQUIVALENCE statements or array specifications were encountered in the subroutines analyzed.

All global constants are initialized in the subroutine INITCT. The subroutine, however, lacks a header or comments to describe the variable names and units of measure for the constants.

FORTRAN '77 does not provide the capability for the user to directly allocate/deallocate memory.

1.3.2 Local Memory

Local memory is accessible only from the currently invoked subroutine. All local variables, whether implicitly defined or not, are deallocated once the subroutine has ended. References to variable space within COMMON blocks are considered references outside the local memory space. Unused variables, or arrays within local memory space, could cause a shortage of memory which might prevent invocation of the subroutine itself. In the modules analyzed, no unused variables or arrays were identified.

Memory allocated as a result of defining and/or using local variables is released every time a subroutine is exited.

1.4 Implications for Model Use

The ALARM code is generally well-structured, with extensive internal comments. It has good modularity, and the logic of the program can usually be easily followed without reference to external documentation. The ALARM source code was analyzed using automated tools and there were few of the problems normally found in codes of this magnitude; e.g., those associated with incorrect parameter passing and COMMON variable mismatches. Since the overall quality of the code is assessed to be good, the user should have confidence that the implementation of sensor functionality in ALARM can be understood.